A Quick Tour of the VeriFast Program Verifier

Bart Jacobs*, Jan Smans, and Frank Piessens

Department of Computer Science, Leuven, Belgium
{bart.jacobs,jan.smans,frank.piessens}@cs.kuleuven.be

Abstract. This paper describes the main features of VeriFast, a sound and modular program verifier for C and Java. VeriFast takes as input a number of source files annotated with method contracts written in separation logic, inductive data type and fixpoint definitions, lemma functions and proof steps. The verifier checks that (1) the program does not perform illegal operations such as dividing by zero or illegal memory accesses and (2) that the assumptions described in method contracts hold in each execution.
Although VeriFast supports specifying and verifying deep data structure properties, it provides an interactive verification experience as verification times are consistently low and errors can be diagnosed using its symbolic debugger. VeriFast and a large number of example programs are available online at: `http://www.cs.kuleuven.be/~bartj/verifast`.

1 Introduction

To tame the problems caused by aliasing when reasoning about imperative programs, O'Hearn, Reynolds and Yang [1, 2] proposed a variant of Hoare logic [3] called separation logic. Separation logic extends Hoare logic with new assertions to describe the structure of the heap. These additional assertions allow for local reasoning through the frame rule:

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{Q * R\}}$$

Informally, the frame rule states that to reason about the behavior of a command C, it is safe to ignore memory locations not accessed by C (here R).

To automate the ideas behind separation logic, Berdine *et al.* [4] proposed an efficient verification algorithm based on symbolic execution and implemented this algorithm for a small, imperative language in Smallfoot. Variants of this algorithm were soon implemented in static analyzers (e.g. Space Invader [5]) and in automatic (e.g. jStar [6]) and interactive program verifiers (e.g. Ynot [7]).

This paper describes the main features of VeriFast, a program verifier that brings the ideas of Berdine *et al.* to (subsets of) C and Java. Contrary to Smallfoot, we focus more on fast verification, expressive power, and the ability to diagnose errors easily than on automation. In the remainder of this paper, we explain the core specification concepts by showing how one can specify and verify full functional correctness of a C implementation of a stack (Section 2), and discuss our experience with our implementation (Section 3).

* Bart Jacobs is a Postdoctoral Fellow of the Research Foundation - Flanders (FWO).

2 Building Blocks

In this section, we introduce the building blocks of the VeriFast approach: method contracts written in separation logic, inductive data types, fixpoint functions and lemma functions. We do so by specifying a C implementation of a stack.

2.1 Method Contracts

In VeriFast, developers can specify the behavior of a C function via a method contract consisting of two assertions, a precondition and a postcondition. Both assertions must be written in a form of separation logic. As an example, consider the program of Figure 1. The function *create_node* creates a new node, initializes its fields, and returns a pointer to the caller. As *create_node* can be called at all times, its precondition (keyword **requires**) imposes no restriction on callers. Its postcondition (keyword **ensures**) guarantees that the fields *value* and *next* of the returned pointer are valid memory locations that respectively hold the values v and nxt. In addition, the conjunct $malloc\_block\_node(result)$ guarantees that the return value is a pointer returned by *malloc*, that can be passed to *free*[1] to deallocate **sizeof**(**struct** *node*) bytes of memory. Note that all aforementioned conjuncts of the postcondition are separated by a separating conjunction (denoted by $*$), indicating that modification of one conjunct will not affect the others.

In Figure 1 and the remainder of this paper, annotations are marked by a gray background. In our implementation annotations must be placed inside special comments that are ignored by the C compiler, but recognized by VeriFast.

```
struct node { int value; struct node * next; };

struct node * create_node(int v, struct node * nxt)
  requires emp;
  ensures result→value ↦ v * result→next ↦ nxt * malloc_block_node(result);
{
  struct node * n := malloc(sizeof(struct node));
  if(n = 0) abort();
  n→value := v; n→next := nxt;
  return n;
}
```

Fig. 1. The function *create_node* and its method contract.

[1] In C, only pointers returned by *malloc* should be passed to *free*.

2.2 Inductive Data Types

To allow developers to specify rich properties, VeriFast supports inductive data types. For example, the first line of Figure 2 defines the well-known inductive data type *list*: a list is either empty, *nil*, or the concatenation of a head element and a tail. Note that the definition is generic in the type of the list elements (here *t*). As we will soon show, inductively defined lists can be used in specifications.

```
inductive list<t> = nil | cons(t, list<t>);

predicate lseg(struct node *f, struct node *t; list<int> vs) =
  f = t ?
    vs = nil :
    f != 0 * f->value |-> ?v * f->next |-> ?n * malloc_block_node(f) *
      lseg(n, t, ?vs0) * vs = cons(v, vs0);

struct stack { struct node *head; };

predicate stack(struct stack *s; list<int> vs) =
  s->head |-> ?h * malloc_block_stack(s) * lseg(h, 0, vs);

struct stack *create_stack()
  requires emp;
  ensures stack(result, nil);
{
  struct stack *s :=
    malloc(sizeof(struct stack));
  if(s == 0) abort();
  s->head := 0;
  return s;
}

void push(struct stack *s, int x)
  requires stack(s, ?vs);
  ensures stack(s, cons(x, vs));
{
  s->head := create_node(x, s->head);
}
```

Fig. 2. A small program illustrating inductive data types and predicates.

To describe recursive data structures and to allow for information hiding, VeriFast supports separation logic predicates. A predicate is a named assertion. For example, Figure 2 defines the predicates *lseg* and *stack*. The former predicate denotes a chain of valid nodes starting at f and ending in t containing exactly the values in the mathematical list *vs*. More specifically, if f equals t, then *vs* is the empty list; otherwise, f is a node with some value v (the question mark preceding v indicates that $f{\rightarrow}value$ can have an arbitrary value, which is called v in the remainder of the assertion), there exists a sequence of node objects at f's next pointer with values *vs0* and *vs* is the concatenation of v and *vs0*. The latter predicate states that s is a valid stack that holds the values *vs*.

The aforementioned predicates are used in Figure 2 to specify the behavior of *create_stack* and *push* in an implementation-independent manner. More specifically, *create_stack* can be called at all times, and guarantees that the returned pointer refers to a valid, but empty stack. The precondition of *push* requires that s is a pointer to a valid stack containing an arbitrary sequence of values called vs. *push*'s postcondition ensures that s still is a valid stack with the value x added at the top.

Both *lseg* and *stack* are precise predicates. This means that their input parameters uniquely determine (1) the structure of the heap described by those predicates and (2) the values of the output parameters. In VeriFast, input parameters are separated from output parameters by a semicolon. For example, f and t are input parameters of *lseg*, while vs is an output parameter. VeriFast automatically tries to fold and unfold precise predicate instances whenever necessary. For instance, the predicate instance $stack(s, vs)$ is opened automatically inside *push* such that $s{\rightarrow}head$ can be read. As shown in Figure 5, developers can insert explicit fold (**close**) and unfold (**open**) proofs steps in the form of ghost commands for non-precise predicates or when the automatic folding and unfolding does not suffice.

2.3 Fixpoint Functions

In addition to inductive data types, VeriFast also supports fixpoint functions. Just like predicates and inductive data types, fixpoint functions can only be mentioned in specifications, not in the C code itself. Figure 3 contains 3 fixpoint functions, that respectively compute the head, tail and length of an inductively defined list. Note that the aforementioned fixpoints functions are generic in the element type of the list.

```
fixpoint t head<t>(list<t> l) {
  switch(l) {
    case nil : return default<t>;
    case cons(hd, tl) : return hd;
  }
}
```

```
fixpoint list<t> tail<t>(list<t> l) {
  switch(l) {
    case nil : return default<t>;
    case cons(hd, tl) : return tl;
  }
}
```

```
fixpoint int length<t>(list<t> l) {
  switch(l) {
    case nil : return 0; case cons(hd, tl) : return 1 + length(tl);
  }
}
```

Fig. 3. The fixpoint functions *head*, *tail* and *length*

The body of a fixpoint function must be a switch statement over one of the fixpoint's inductive arguments. To ensure soundness of the encoding of fixpoints, VeriFast checks that fixpoints terminate. In particular, VeriFast enforces that whenever a fixpoint g is called in the body of a fixpoint f that either g appears before f in the program text or that the call decreases the size of an inductive argument. For example, the call $length(tl)$ in the body of *length* itself is allowed because tl is a component of l (and hence smaller than l).

As shown in Figure 4, *pop*'s function contract uses fixpoint functions: given a non-empty stack with values *vs*, *pop* removes the top of the stack (i.e. the head of *vs*) and returns this value to the caller. The function *dispose* deallocates a stack and its constituent nodes. To dispose the nodes, *dispose* walks over the list of nodes in a loop and deallocates them one by one. To reason about loops, VeriFast requires developers to provide loop invariants (keyword **invariant**). Developers may provide an optional loop variant, an integer-valued expression that decreases in each iteration but never becomes negative, to enforce termination. In the example, the length of the sequence of nodes that is not deallocated yet is the loop variant.

```
int pop(struct stack * s)
  requires stack(s, vs) * vs ≠ nil;
  ensures stack(s, tail(vs))*
          result = head(vs);
{
  int r := s→head→value;
  struct node * n := s→head;
  s→head := n→next;
  free(n);
  return r;
}
```

```
void dispose(struct stack * s)
  requires stack(s, ?vs);
  ensures emp;
{
  struct node * n := s→head;
  while(n ≠ 0)
    invariant lseg(n, 0, ?vs0);
    decreases length(vs0);
  {
    struct node * tmp := n→next;
    free(n); n := tmp;
  }
  free(s);
}
```

Fig. 4. The functions *pop* and *dispose*.

2.4 Lemma Functions

Lemma functions allow developers to prove properties of their inductive data types, fixpoints and predicates, and allow them to use these properties when reasoning about programs. A lemma is a function without side-effects marked **lemma**. The contract of a lemma function corresponds to the property itself, its body to the proof and a lemma function call corresponds to an application of the property. VeriFast has two types of lemma functions, pure and spatial lemmas.

```
fixpoint list<t> append<t>(list<t> a, list<t> b) {
  switch(a) {
    case nil : return b; case cons(hd, tl) : return cons(hd, append(tl, b));
  }
}

lemma void append_assoc<t>(list<t> a, list<t> b, list<t> c)
  requires true; ensures append(append(a, b), c) = append(a, append(b, c));
{ switch(a) { case nil : ; case cons(hd, tl) : append_assoc(tl, b, c); } }

lemma void lseg_add(struct node * a)
  requires lseg(a, ?b, ?vs1) * b→next ↦?n * b → value ↦?v*
    malloc_block_node(b) * lseg(n, 0, ?vs2);
  ensures lseg(a, n, append(vs1, cons(v, nil))) * lseg(n, 0, vs2);
{
  if(a = b){ open lseg(a, b, vs1); } else { lseg_add(a→next); }
  open lseg(n, 0, vs2); close lseg(n, 0, vs2); // get info from predicate body
}

int size(struct stack * s)
  requires stack(s, ?vs); ensures stack(s, vs) * result = length(vs);
{
  int c := 0; struct node * n := s→head; struct node * head := n;
  while(n ≠ 0)
    invariant lseg(head, n, ?vs1) * lseg(n, 0, ?vs2)*
      c = length(vs1) * vs = append(vs1, vs2);
    decreases length(vs2);
  {
    c++; n := n→next;
    lseg_add(head); append_assoc(vs1, cons(head(vs2), nil), tail(vs2));
  }
  return c;
}
```

Fig. 5. The correctness of the C function *size* is established using the lemma functions *lseg_add* and *append_assoc*.

A pure lemma is a function whose contract only contains pure assertions, and whose body proves that the precondition implies the postcondition. *append_assoc* shown in Figure 5 is a pure lemma that proves by induction on a's size that the fixpoint *append* is associative[2]. More specifically, the case *nil* of the switch statement corresponds to the base case, while the case *cons* corresponds to the inductive step. Note that switches over inductive data types do not require break statements.

As opposed to pure lemmas, contracts of spatial lemmas can mention spatial assertions such as predicates and points-to assertions. A spatial lemma with precondition P and postcondition Q states that the program state described by P is equivalent to the state described by Q. A spatial lemma call does not modify the underlying values in the heap, but changes the symbolic representation of the program state. *lseg_add* shown in Figure 5 is an example of a spatial lemma that shows that a list segment from a to b can be extended provided b itself is a valid node. The body of the C function *size*, which computes the number of elements in a stack s, calls *lseg_add* and *append_assoc* to prove that the loop invariant is preserved by the loop's body.

3 Implementation and Experience

The VeriFast program verifier, a large number of examples, and additional documentation is available online at: `http://www.cs.kuleuven.be/~bartj/verifast`. VeriFast has been used for teaching several courses on program verification at K.U.Leuven (Belgium) and ETH Zurich (Switserland). The documentation includes a tutorial, which describes the supported subset of C via a number of examples and covers many features of VeriFast not discussed here such as fractional permissions, higher-order predicates, overflow checking, function pointers, predicate families and concurrency.

VeriFast has been used in a number of case studies as shown in the table below. These case studies do not consist of large code bases, but rather focus on proving correctness of challenging specification and verification patterns (e.g. composite).

program	total # lines	# annotation lines	time taken (seconds)
chat server	242	114	0.08
linked list and iterator	332	194	0.09
composite	345	263	0.09
JavaCard applet	340	95	0.51
GameServer	383	148	0.23

To make it easier for developers to diagnose verification errors, VeriFast has an IDE that supports symbolic debugging. That is, when verification fails, one can inspect the symbolic states encountered during symbolic execution on the path to the failure. A screenshot of the IDE is shown in Figure 6.

[2] Our pure prover, Z3 [8], does not perform induction and therefore cannot derive associativity of *append* solely based on its definition.

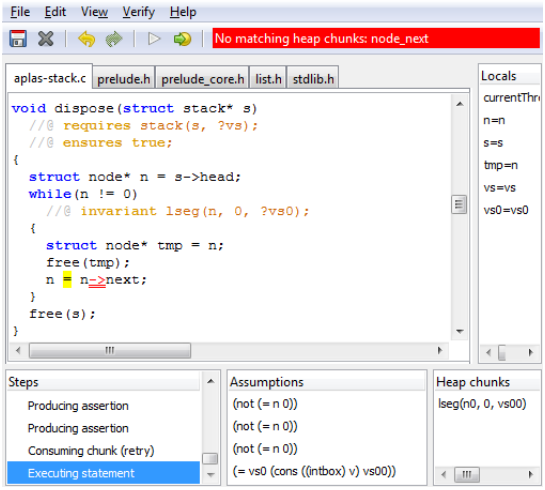

Fig. 6. A screenshot of the VeriFast IDE. Developers can use the symbolic debugger in the IDE to diagnose verification errors and inspect the symbolic state at each program point. The box on the bottom left of the screen shows the symbolic states encountered on the current path. The components of the selected state are shown in the boxes on the bottom center (path condition), bottom right (symbolic heap), and top right (symbolic store).

References

1. Peter O'Hearn, John Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. *CSL*, 2001.
2. John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Symposium on Logic in Computer Science*, 2002.
3. C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12, 1969.
4. Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Symbolic execution with separation logic. In *APLAS*, 2005.
5. Cristiano Calcagno, Dino Distefano, Peter W. O'Hearn, and Hongseok Yang. Compositional shape analysis by means of bi-abduction. In *POPL*, 2009.
6. Dino Distefano and Matthew Parkinson. jStar: Towards practical verification for Java. In *OOPSLA*, 2008.
7. Aleksandar Nanevski, Greg Morrisett, Avi Shinnar, Paul Govereau, and Lars Birkedal. Ynot: Reasoning with the awkward squad. In *ICFP*, 2008.
8. Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.